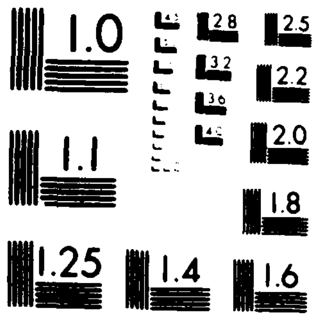

1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

(1)

A SPECIFICATION TECHNIQUE FOR THE COMMON APSE INTERFACE SET*

by

Timothy E. Lindquist
Jeffrey L. Facemire
Dennis G. Kafura

Computer Science Department

Virginia Polytechnic Institute
and State University
Blacksburg, Virginia 24061

A SPECIFICATION TECHNIQUE FOR THE COMMON APSE INTERFACE SET*

by

Timothy E. Lindquist
Jeffrey L. Facemire
Dennis G. Kafura

Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

March 20, 1984

----

* This research was supported by the Ada Joint Program Office througn the Office of Naval Research Information Sciences Division under ONR contract number N00014-83-K-0643. The effort was under the technical direction of Virginia L. Castor Wright-Patterson AFB, Ohio. Reproduction in whole or in part is permitted for any purpose of the United States Government. The views expressed herein are solely those of the authors.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 84004-R	2. GOVT ACCESSION NO. AD-A240	3. RECIPIENT'S CATALOG NUMBER 889
4. TITLE (and Subtitle) A Specification Technique for the Common APSE Interface Set		5. TYPE OF REPORT & PERIOD COVERED Technical
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Timothy E. Lindquist, Jeffrey L. Facemire, Dennie G. Kafura		8. CONTRACT OR GRANT NUMBER(s) N00014-83-K-0643
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department, VPI & SU, Blacksburg, VA 24061		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research, Code 442, 800 North Quincy St., Arlington, VA 22217		12. REPORT DATE April, 1984
		13. NUMBER OF PAGES 38
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) AFWAL/AAAF, Wright-Patterson AFB, Ohio 45433		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approval for public release, distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada, Programming Environments, APSE, CAIS, Specification, Abstract Machines, Validation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report demonstrates an approach to specifying kernel Ada support environment interface components. The objectives are to provide a mechanism which allows building a complete enough specification for validation, an understandable specification, and one that is relatively easy to construct. In meeting these objectives, an Abstract Machine approach has been modified and applied to functional description of kernal operations. After motivating and explaining the approach, the paper exemplifies its utility. (over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Interactions among kernal operations and pragmatic implementation limits, which are other needed parts of a specification, are also discussed.

ABSTRACT

This report demonstrates an approach to specifying kernel Ada support environment interface components. The objectives are to provide a mechanism which allows building a complete enough specification for validation, an understandable specification, and one that is relatively easy to construct. In meeting these objectives, an Abstract Machine approach has been modified and applied to functional description of kernel operations. After motivating and explaining the approach, the paper exemplifies its utility. Interactions among kernel operations and pragmatic implementation limits, which are other needed parts of a specification, are also discussed.

CONTENTS

I. PROBLEM STATEMENT

A fundamental goal of the Ada* program is to increase portability between different Ada Programming Support Environments (APSEs). Recognizing that transportability extends beyond language issues, the Ada Joint Program Office has formed the Kernel APSE Interface Team and Kernel APSE Interface Team for Industry and Academia (KIT and KITIA) [8]. These teams are formulating the requirements for, and preliminary form of, a set of kernel interfaces for tools that are needed to support an APSE. The thesis driving this development is: if programs that comprise an APSE use the same interface to the underlying kernel then tools and data will be transportable among APSEs. The KIT and KITIA have designed a preliminary version of the interfaces necessary for supporting program development tools. The interfaces [2], called the Common APSE Interface Set (CAIS--pronounced as case), extends the functionality of Ada as needed for implementing APSE tools. As the name implies, CAIS is to be adopted on all Ada development environments as the interface between tools and their underlying kernel facilities.

Tool transportability can be viewed as having three necessary conditions. The language processor must be identical

----

*Ada is a registered trademark of the United States Department of Defense Ada Joint Program Office.

across APSEs, tool-to-tool protocols must be identical, and the tool-to-kernel interface must be identical. Figure 1 shows the three types of interactions that correspond to these conditions. Tool-to-tool protocols are represented with a dashed edge indicating an indirect interaction. Any communication one tool has wit`` another is actually realized through either the language oı ‹ernel interface. Another example of an indirect interaction, although not involved in transportability, is the human-to-tool interface. User interactions with a tool are also realized indirectly through underlying services. The Ada Compiler Validation Center (ACVC) addresses the first transportability condition by supporting and administrating a Compiler Validation Suite [9]. The second condition requiring the use of common tool-to-tool protocols refers to interactions among tools. Tool-to-tool protocols may take the form of intermediate files, message streams, inter-tool timing and synchronization, or resource contention. When transporting an isolated tool, independent of others with which it communicates, these protocols must be addressed. For instance, if a debugger were to be transported from one APSE to another, the compiler-to-debugger protocols, such as those manifest through the symbol table and intermediate code, must be identical on the original and target APSEs.

The interface to kernel operations, which support files, devices, and processes, must also be identical to transport a tool. KIT and KITIA have designed the CAIS to address this aspect of transportability. One of the objectives of the APSE

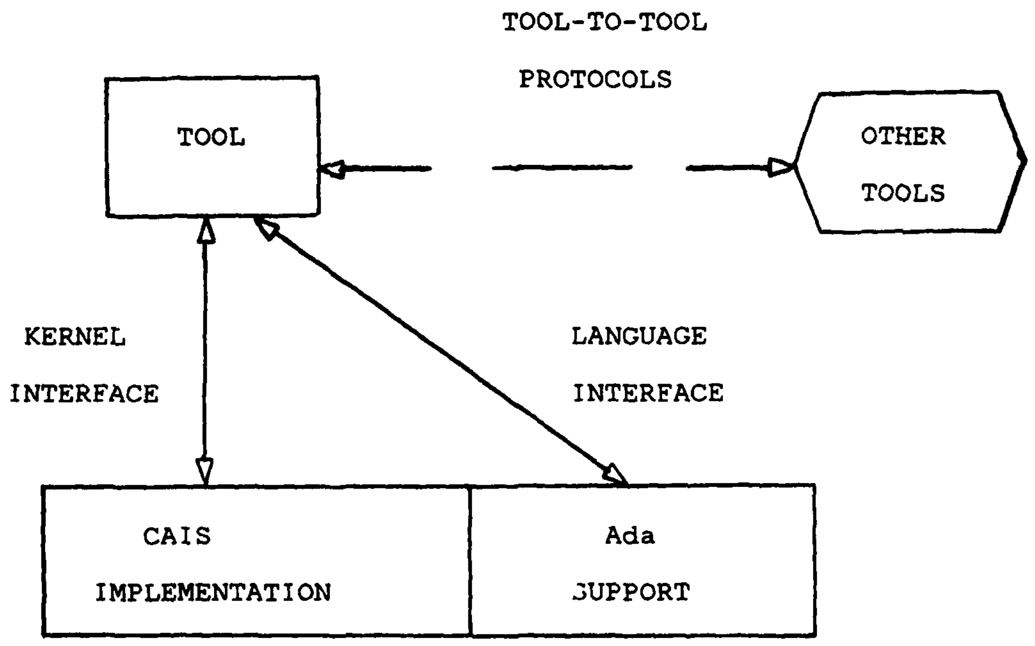


Figure 1. APSE Tool Transportability Interfaces.

Evaluation and Validation Team (E&V) is to initiate the development of a CAIS Validation Suite [1]. The suite will be administered to assess compliance to the CAIS specification in much the same way as the Compiler Validation Suite is used. If a tool uses only CAIS facilities and uses them in a manner consistent with the CAIS specification then this transportability requirement is satisfied. While functional compatibility of CAIS implementations will be assured through validation, a tool will not transport unless it uses CAIS according to specifications. For example, a tool may be written to depend upon functionality that extends the CAIS in a way that is not detectable by the validation suite.

A prerequisite to developing the CAIS Validation Suite is a clear specification. Since the purpose of validation is to assess the degree to which an implementation adheres to the specification, a concise, complete, and consistent specification is needed. Aside from its utility to users and implementors, the specification is the primary input to constructing a validation suite.

An example from CAIS further motivates this need and demonstrates how difficult complete natural language specifications are to write. CAIS section 6.2.2 describes synchronous and asynchronous operations that allow one process to call another. In describing the parameter RELATIONSHIP_KEY to the asynchronous call, CAIS states:

> "The calling task can either supply the KEY or the CAIS implementation will assign a key via UNIQUE_CHILD_KEY."

Although the statement itself is clear, when taken in the context of other CAIS operations it is either incomplete or inconsistent. According to the statement, the user-defined key could be a duplicate (the same key as another child's of the same primary relation). If this is indeed allowed, it is inconsistent with other CAIS operations that don't specify the meaning of accessing a duplicate key. More likely, the statement is incomplete, and the intended meaning is:

> If the calling task supplies a key that is unique to all siblings of the same relation then that key is used. Otherwise, the implementation will assign a key that is unique across the relation.

An incomplete or inconsistent specification leaves the validator with the choice of either ignoring a potentially distinguishing semantic characteristic or interpreting the intentions of the designer. As applied to the child key example, the validator could either not test a duplicate key or assume the designers intent to force no duplicates.

The report of a preliminary study of validation in an APSE [7], indicates that specifying an interface set, such as CAIS, requires more than a description of the syntax and functionality. Additionally, interactions that exist at the interface must be specified. Hidden within the implementation, operations are related to each other in the same way as tools are related. Several CAIS operations may use a common data structure or may be synchronized. Further, any pragmatic limits which apply to implementations must also be specified. These might include the length of identifier strings, field sizes, maximum number of processes, or maximum number of times a facility can be called.

The question addressed in this report is: How can the meaning of CAIS be specified in a manner that is readable, lends itself to the complete capture of semantics, and aids in constructing a validation suite? The remainder of the report discusses a CAIS specification technique that incorporates separate parts for functionality, interactions, and pragmatic limits. The syntax of CAIS operations is also a necessary part of the specification, but it is assumed that the visible part of Ada package specifications are a good vehicle for this description.

II. SPECIFYING FUNCTIONALITY

CAIS consists of several package specifications, some defining types and others defining interface operations. Package specifications, however, only provide syntactic structure. This structure distinguishes between procedures and functions, the names and types of parameters, and return types. Currently, the functionality of operations is specified through commentary descriptions. For the purposes of this paper, functionality means the manipulations and checks made on input arguments, any conditions causing errors/exceptions, specific return status, and the outputs (effects) generated by an operation.

The validation report [7] discusses four alternative methods for describing interface functionality including English commentary, examples, formal descriptions, and Abstract Machines.

Commentary, or a natural language description, is currently being used in the CAIS specification since it is easy to construct and comprehensible, but it is easy to overlook key semantic issues, which often result in incompleteness or ambiguity. When presented with commentary descriptions, the validator is often left with the task of resolving ambiguity and making arbitrary semantic decisions as indicated in the example of a child key already cited. While it is true that incomplete specifications can be generated using any technique, the more rigorous structure of formal and Abstract Machine descriptions make incompleteness less of a problem than with commentary. Natural language descriptions of functionality are valuable during the design phase, but must be replaced by increasingly more complete techniques as the design solidifies.

One method of specifying the entire CAIS semantics, which has been explored by Freedman [3], uses a formal technique based on Denotational Semantics [10]. Formal specifications could use either a Denotational or Axiomatic [6] approach, which provide the mathematical meaning of each of the operations being specified. If an axiomatic approach were adopted, a mathematical theory of CAIS would be developed. Axioms (or assumed truths) would be designed, in the form of logical statements. Each axiom would describe the functionality of an operation such as INVOKE_PROCESS. For example an axiom for OPEN might take the form:

PRE {OPEN(path,handle)} POST

Where PRE and POST are logical propositions. The interpretation of the axiom states that if PRE is satisfied before invoking OPEN then POST is satisfied when (if) it completes. PRE and POST, describe in terms of path and handle, the functionality of OPEN. Rules would also be constructed showing how axioms could be combined to describe the meaning of invoking various combinations of operations. Theorems of the theory could then be proven indicating the meaning of various CAIS operations as they might be used by a tool.

Functionality can also be described operationally in the form of Abstract Machines. To apply this approach, programs are written to describe what CAIS operations do. If there existed an executor for the programs (the Abstract Machine) then an operational definition of CAIS functionality would exist. Freedman indicates that an operational description of programming languages (usually a prototype compiler) has traditionally existed prior to a formal description. In a later point paper [4], he suggests that such a description of CAIS be developed. Abstract Machines can provide an adequate link between the operational and formal definitions providing that the two primary drawbacks discussed by Freedman [3] are resolved.

One drawback is that the Abstract Machine approach is bottom-up. Before descriptions of operations can be detailed (or understood), the instructions and data recognizable by the machine must be designed. It is true that any semantic description technique must define the meta-language in which the description is formulated. For the CAIS (its users and implementors), the best solution to this problem is to describe functionality in an Ada-like language, which is done in the Abstract Machines presented in this paper.

The second drawback to Abstract Machines is that they may bind the implementor to a specific implementation technique. While an Abstract Machine description of a CAIS operation can indicate a specific implementation, the technique used to construct Abstract Programs can minimize implementation dependence. For example, one abstract description of CAIS Node Handles might completely specify their type. Implementation independence can be retained by incompletely specifying the handle and only indicating the primitive operations that the Abstract Program needs to perform on a handle. Beyond properly constructing abstract programs, it must be emphasized that the descriptions generated as Abstract Programs define functionality only through the effects that their conceived execution would have. Although Abstract Machine descriptions may implicate a single implementation, any implementation generating the same externally observable effect is suitable.

A. ABSTRACT MACHINES

This section characterizes the Abstract Machine approach, and presents an example Abstract Program for a part of CAIS Process Control. Writing programs for an Abstract Machine, which describe CAIS operations, is the basis for this approach. The utility of such a description is independent of an actual executor for the machine. The value of the technique depends greatly on whether the intended audience can understand the meanings of Abstract Programs. For CAIS, an Abstract Program describing INVOKE_PROCESS is only useful if CAIS users, designers, and implementors can understand the operation independent of an executor. Since the Abstract Machine may never be built, it must be well-defined and human understandable.

The Abstract Machine, for which programs are presented in this paper, consists of three components:

1. A Processor
2. A Storage Facility
3. An Instruction Set

The processor is able to recognize and execute instructions from a predefined set. Each instruction has an action that the processor carries out in some data context. One component of the processor, called the environment pointer, indicates the data context in which an instruction is to be executed.

Another component, the instruction pointer, allows the processor to sequence through the instructions of the program executing them as appropriate.

The storage of the processor is a memory for both data and programs. Data values, as named through program identifiers, may be stored for reference throughout the execution of a program. The last component of the Abstract Machine, the instruction set defines actions that may be performed in a program. The foundation of the instruction set used in this report is Ada.

The functionality of a CAIS operation is given by detailing an Ada-like package body. Package bodies are described through a set of procedural instructions, which if executed would perform the intended function. In constructing an Abstract Program, operations are needed that are not part of the Ada language. These operations and primitive objects, which are also needed to augment Ada, are treated as axioms of the machine. The meanings of primitive objects and operations are left to commented package specifications. These additional packages, whose bodies are not detailed, can be viewed as extending the instruction set of the Abstract Machine to include operations, objects, and types beyond the scope of Ada. Figure 2 illustrates the Abstract Machine for CAIS. The processor and storage have components for the Ada language as well as additional aspects indicating the ability to extend Ada. Abstract Programs are indicated to exist for each CAIS

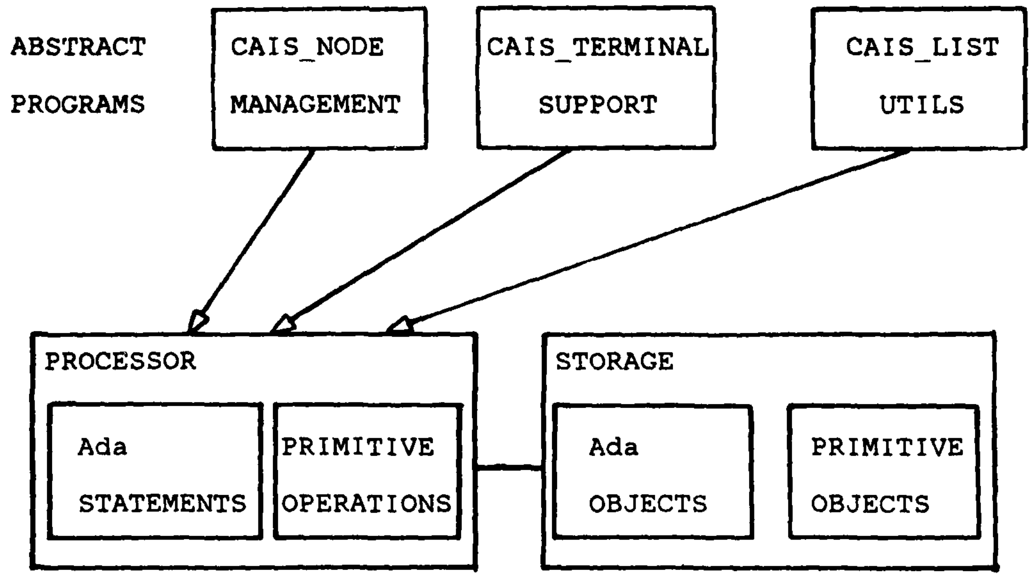


Figure 2. Abstract Machine for CAIS Functionality

package defined.

Several reasons make Ada a desirable vehicle for the definition. One reason is the richness of Ada control constructs and type facilities, which are enhanced by packages, exceptions, and tasking. These make it particularly suitable for semantic description. Further support for using Ada lies in the observation that any language used as a semantic description tool must have well-defined semantics of its own. Although Ada has not been specified formally, the language's controlled definition does provide an adequate semantic base for the Abstract Machine. Another reason for using Ada is that the descriptions can be compiled, but the most compelling reason is that it is compatible with the problem and user environment. CAIS implementors and users will be familiar with Ada making the Abstract Programs much more comprehensible.

B. THE ABSTRACT PROGRAM FOR SPAWN_PROCESS

The CAIS specification currently consists of a group of related Ada package specifications together with associated commentary. The package specifications demonstrate the syntax of operations, and English commentary is used to describe both the semantics and rationale for components. The Process Control package of CAIS provides for invocation, state query, temporary suspension, and termination of programs. Using a process for each Ada program, a tree structure of process nodes exists for each job. The root of the tree is called CURRENT_JOB. Each time a process invokes another, the new process becomes a child of the caller. All processes in the tree are uniquely identified by a pathname that consists of a sequence of delimited pairs of relation names and relationship keys. Each pair in the pathname corresponds to a single level of the tree. A child is uniquely selected by the pair, where the relation name (DOT--abbreviated as '.') identifies among possibly many relations emanating from a node, and the key identifies one of possibly many siblings of the relation.

Mechanisms provided for invoking a program include synchronous and asynchronous calls. The synchronous facility is the logical equivalent of procedure call. The caller transfers control to the called program and awaits its completion, at which time the caller continues execution. The procedure INVOKE_PROCESS implements synchronous calls and includes parameters that return the results and completion status of the called program. Asynchronous call (SPAWN_PROCESS) initiates a program as requested, but provides no further communication or synchronization. Once the new program has been initiated, the caller continues independent of the called program. No feedback to the caller is automatically provided, but the operation AWAIT_PROCESS can be called to synchronize with completion and obtain results. Communication among asynchronously executing programs can be accomplished through the message operations provided by the package CAIS_PROCESS_COMMUNICATION.

Figure 3 contains an Abstract Program for the asynchronous calling mechanism SPAWN_PROCESS, which is contained in the Abstract Program for CAIS_PROCESS_CONTROL. Although the remainder of the body is not shown, it defines the context for SPAWN_PROCESS to include the necessary node and process definitions. The Abstract Program for SPAWN_PROCESS causes a process node to be created for the program and causes the process to begin execution. The indicated program is first found and verified to be in executable form. By opening the associated file node, the pathname is traversed obtaining a

```
procedure SPAWN_PROCESS  (PROGRAM: in PROGRAM_STRING;
     PARAMS: in PARAMS_STRING; NODE: in out NODE_TYPE;
     KEY: in out RELATIONSHIP_KEY:=UNIQUE_CHILD_KEY;
     STD_IN: in FILE_TYPE:=CAIS_TEXT_IO.CURRENT_INPUT;
     STD_OUT: in FILE_TYPE:=CAIS_TEXT_IO.CURRENT_OUTPUT;
     STD_ERROR: in FILE_TYPE:=CAIS_TEXT_IO.CURRENT_ERROR;
     CURR_NODE: in NAME_STRING:="'CURRENT_NODE");

   IS_UNIQUE : BOOLEAN:=TRUE;
   NODE,NEXT_NODE : NODE_TYPE;
   FILE_TYPE : CAIS_LIST_UTIL.LIST
   ITERATOR : NODE_ITERATOR;

  begin

   OPEN(NODE,PROGRAM);
   if KIND(NODE) /= FILE
      then CLOSE(NODE);
           raise NAME_ERROR;
   end if;
   GET_NODE_ATTRIBUTE(NODE,"file_type",FILE_TYPE)
   if CAIS_LIST_UTIL.IDENTIFIER(FILE_TYPE)/='executable_image';
      then CLOSE(NODE);
           raise NAME_ERROR;
   end if;
   CLOSE(NODE);
--Assume CURRENT_PROCESS is a handle on myself.
--Determine whether the user specified key is unique.
   ITERATE(ITERATOR,CURRENT_PROCESS,KIND=>PROCESS);
   while (IS_UNIQUE and MORE(ITERATOR)) loop
      GET_NEXT(ITERATOR,NEXT_NODE);
      if PRIMARY_KEY(NEXT_NODE) = KEY
         then KEY := UNIQUE_KEY(CURRENT_PROCESS,'.',KEY);
              IS_UNIQUE:=FALSE;
      end if;
   end loop;
   CREATE_PROCESS_NODE(CURRENT_PROCESS,KEY,PROGRAM,PARAMS,
           'ready',STD_IN,STD_OUT,STD_ERR);
   CONCURRENT_RUN(CURRENT_PROCESS,'.',KEY);
 end SPAWN_PROCESS;
```

Figure 3. Abstract Program for SPAWN_PROCESS

handle to the program. A check is then made to assure that the file node contains an executable program. The argument list to SPAWN_PROCESS includes a relationship key to be used in naming the newly created process node. SPAWN_PROCESS checks to see whether the key is unique among other processes already initiated. The final actions taken by SPAWN_PROCESS are to create the process node as a direct descendant of the caller and to request execution of the new process.

In current form, CAIS does not adequately address valid inputs or error/exceptional conditions. As an example, the first parameter to both forms of program invocation is the name of the program being invoked. Although it is intuitively clear what a program name is, this parameter must be further defined. Syntactically, aside from the fact that the program name is a string, what is the proper form for a program name? Are there special characters that may or may not be allowed in a program name? In this case, the intention is that the name must be a valid file system name. Are there any additional constraints on the name as is often the case in interactive systems (eg, .EXE suffix)? What happens when the name is not syntactically correct? A further line of questioning revolves around the existence of the program, privileges to access it, and request its execution. The answers to many of these questions can be provided by reference to other CAIS components. For example, the syntactic form of a pathname is detailed in CAIS Section 3.1.3. Figure 3 references OPEN from CAIS_NODE_MANAGEMENT to provide name validation. Such references allow duplication of semantic description to be avoided.

The way that CAIS operations handle errors and the conditions causing errors can be made clear through Abstract

Programs. CAIS provides exceptions to indicate to the caller such occurrences as NAME_ERROR, USE_ERROR, and CAPACITY_ERROR. Abstract Programs for operations that raise exceptions can indicate under what conditions the exception is raised and whether any other CAIS components handle the exception (supposing an exception is raised in a CAIS procedure called by a CAIS procedure).

The statements:

```
if CAIS_LIST_UTIL.IDENTIFIER(FILE_TYPE)/='executable_image'
  then CLOSE(NODE);
       raise (NAME_ERROR);
end if;
```

make it clear that the condition causing the NAME_ERROR exception is that the PROGRAM argument does not name an executable node. The absence of a handler, indicates that the caller of the facility is responsible for deciding on an appropriate action.

SPAWN_PROCESS requires a unique key as an argument that identifies the process being created. If the user wishes to spawn a program as the child of the current process, but does not know a unique key for the new process, then the CAIS facility will generate one. While it is indeed important that all keys for the descendants of a process node be unique, it would be desirable to have the CAIS force that uniqueness by changing the argument KEY provided by the user. In Figure 3, the KEY parameter has been changed to IN OUT, and SPAWN PROCESS checks for the uniqueness of the KEY. If it is not unique then a unique one is obtained as the new value of KEY. This is an example in which there are two distinct successful completion states for a CAIS facility. Using an additional parameter to return status information pertaining to the execution of the facility is much easier in this instance than using exceptions.

In the Abstract Program for SPAWN_PROCESS, interactions with (uses of) other CAIS operations are made explicit by inclusion of the calls to those facilities. Examples of this are the uses of OPEN, CLOSE, KIND, GET_NODE_ATTRIBUTE, ITERATE, and MORE which are Node Model routines; and IDENTIFIER which is from CAIS List Utilities. Two operations are called, however, which are not defined elsewhere in CAIS. CREATE_PROCESS_NODE is used to build and initialize storage for a process. CONCURRENT_RUN is used to indicate that once a process node has been created and properly initialized, that something is done to allow execution. No specific details, aside from a package specification including them, are given for these operations. They are assumed to be operations executable by the Abstract Machine. Commentary in the specification of CONCURRENT_RUN might describe its functionality as:

> The newly created process will begin execution concurrently with the current process. Whatever action that causes the process to complete will be reflected in the STATE and COMPLETION_STATUS attributes of this process node.

III. SPECIFYING PROTOCOLS AND HIDDEN INTERFACES

CAIS defines an interface providing kernel services to program development tools. Operations alone characterize this interface as it appears to the tool writer, but in an implementation of CAIS, interactions take place among operations and with the environment encompassing CAIS. Specifying the functionality of operations only partially exhibits these interactions, which are termed Protocols and Hidden Interfaces. In this section we categorize these interactions, indicate why they are important to validation, and indicate how they can be specified.

A. HIDDEN INTERFACES

The distinction between Protocols and Interfaces is based on the application of the Open Systems Interconnection (OSI) Model to an APSE as detailed in [7] and later expanded and refined by Goodwin [5]. In one form of this model, the APSE consists of layers of logical levels as shown in Figure 4. One level is made up of development tools, another below it is the CAIS, and below CAIS is all that is needed to support CAIS.

```
                     Peer-to-Peer Protocols
                +---------+---------+---------+
  Application-->|         |<------->|         |<--APSE
                +---------+---------+---------+
 Presentation-->|         |<------->|         |<--Transfer
                +---------+---------+---------+
      Session-->|         |<------->|         |<--CAIS
                +---------+---------+---------+
    Transport-->|         |<------->|         |)
                |         |         |         |)
      Network-->|         |<------->|         |)
                |         |         |         |)<--O.S./
    Data Link-->|         |<------->|         |) hardware
                |         |         |         |)
     Physical-->|         |<------->|         |)
      +---------+---------+---------+---------+---------+
      |                  Physical Media                 |
      +-------------------------------------------------+
```

Figure 4. APSE Reference Model

Using this model, Hidden Interfaces are the interactions that provide for communication between objects at different levels. An implementation of the CAIS has two interfaces, one with the tools that use CAIS operations, and the other with the underlying operating system or runtime system. The word Interface in "Common APSE Interface Set" refers to the tool interface.

Upward Hidden Interfaces, those with tools, are the rules that detail how CAIS operations may be used. These rules further the interface specification by crystalizing the functional interdependencies among CAIS operations. In almost every instance, the result of one CAIS operation can only be viewed through another. For example, if one process uses the SEND operation to communicate with another, a corresponding

RECEIVE must be invoked to obtain the information. As another example, before any operations may be performed on a node, OPEN must be invoked to obtain a valid handle. Upon completing operations on a node, the handle must be nullified using CLOSE (or UNLOCK). In general terms, Abstract Programs detail what CAIS operations do, and Hidden Interfaces augment this information with the rules governing how CAIS operations interact when called by tools.

The interactions are important to the tool writer, but are also needed to construct a validation mechanism. Since the implementation details of a particular CAIS will be hidden from the validation tool, it must determine proper functionality of an operation by observing through another operation. For example, OPEN can only be validated by using the generated handle in other CAIS operations, and by observing response to erroneous input. One form of OPEN demonstrates this in more detail.

```
OPEN(NODE: in out NODE_TYPE, NAME: in NAME_STRING)
```

When calling OPEN an output is generated in the parameter NODE that is a handle to the object specified by the pathname NAME. A validation suite is unable to examine the details of NODE to determine whether OPEN works correctly. Instead the suite must use OPEN to generate a handle and then exercise the correctness of the handle through other CAIS operations.

Another example of an Upward Hidden Interface is the attribute attached to file nodes indicating whether the file contains an executable program. This specific interface exists between SPAWN_PROCESS and the APSE tool that creates the file node (Linker). While it is not important that all implementations of SPAWN_PROCESS use the same IDENTIFIER list shown in Figure 3, it is important that a file node contain an indication of its contents that can be examined by SPAWN_PROCESS. The validation suite must exercise this protocol.

Downward Hidden Interfaces are the rules and conventions governing interactions between a CAIS implementation and the underlying operating system. One such interaction exists between CAIS and the runtime support for Ada. If CAIS operations are to raise, propagate, and possibly handle exceptions then implementations must follow the same conventions as the Ada runtime system. CAIS operations, which may or may not be implemented in Ada, must be able to raise exceptions and cause them to be propagated to the calling tool. Any operation that uses another CAIS operation must be able to either handle or propagate exceptions generated by the called routine. A CAIS implementation may either use existing services for treating exceptions or may follow specified conventions that implement exception semantics. In either situation, the validation of a CAIS implementation needs to fully exercise the interface to tools where exceptions are concerned.

In general, Downward Hidden Interfaces exist when the CAIS implementation uses an underlying object that is also used by tools. In the example cited above, the details of exception implementation need to be common to both CAIS and APSE tools. This form of interaction has, in part, motivated a position by Gargaro* emphasizing that runtime support needs to be addressed as part of the CAIS (Common APSE Interface Set). Independent of the common runtime support issue, downward Hidden Interfaces need to be specified because of their impact on implementation and validation. For the implementor, the specification must detail what existing services must be used, or must detail what conventions must be adhered to. For the validator, the specification guides forming tests exercising either proper use of existing services or adherence to conventions.

B. PROTOCOLS

Protocols refer to communication that takes place between other objects at the same level. Protocols govern interactions among CAIS operations, and are specified through the Abstract CAIS Programs. As was pointed out in a previous section, a

----

*Anthony Gargaro KIT/KITIA Position developed based on the paper, Program Invocation and Control in KAPSE Interface Team Public Report Volume II, NOSC TD522, NOSC, San Diego CA, pp.3L1-3L7, October 1982.

functional description of one operation may include calls to other CAIS operations. This type of interaction, called Uses-Protocol, shows a functional hierarchy within CAIS. As an example, SPAWN_PROCESS (Figure 3) uses Node Management OPEN to obtain a handle to the file node containing the program to be executed.

A previous section has stated that the instructions within an Abstract Program do not limit implementations. The functionality of an operation is not defined by the instructions of the program, but is defined by the effect of executing the program on the Abstract Machine. The question arises, however, whether implementations of SPAWN_PROCESS should use OPEN in the same manner as shown in the Abstract program? The advantage of requiring the use of OPEN is that the number of test cases needed to validate SPAWN_PROCESS would be greatly reduced. For example, several different input conditions for OPEN may result in raising the EXCEPTION NAME_ERROR. If the validation suite for OPEN tests each of these input conditions then only two separate corresponding cases are needed for SPAWN_PROCESS. Relying on the fact that OPEN is called and has already been validated, SPAWN_PROCESS need only be tested to assure that it acts appropriately for both exceptional and normal returns from OPEN. If, however, SPAWN_PROCESS is not required to use OPEN, then a test case must be generated for each possible syntactic error and nonexistent node error that may cause SPAWN_PROCESS to propagate a NAME_ERROR. While requiring CAIS operations to use others has clear advantages and disadvantages, including Use Protocols in Abstract Programs displays a needed functional hierarchy of CAIS.

IV. SPECIFYING PRAGMATIC LIMITS

The final part of a CAIS specification provides details of any limits which are applicable to the operations being specified. One type of limit provides a bound on the use of an operation or object. Another provides a limit on the size of a CAIS object, such as identifier strings, number of entries, and length of message strings. Use Limits specify constraints on the control structure of tools. For example, the number of processes that may be spawned by another or the number of message channels that a process uses are determined by the sequence in which instructions in the tool are executed. Value limits on the size of CAIS objects, however, affect arguments used in calls to CAIS operations. Both types of limits need to be specified.

While the distinction between Use limits and Value Limits is not important for the CAIS user or implementor, each present different problems to transporting tools. For example, consider the limits on channels from Section 6.6 of CAIS [2].

> A conforming implementation must support channel names of up to 20 characters. A conforming implementation must support up to 20 simultaneous accepting channels from the same process.

Channels, which are used to communicate messages between processes, are limited in both the size of their names and in the number a process may use at a given time. Limiting the number of characters in a channel name is an instance of a Value Limit, while limiting the number of simultaneous channels is a Use Limit. The task of validating that a CAIS implementation conforms to the limits can be done in a straightforward manner. A validation tool can exercise CAIS operations both within and outside the limits specified. The question of whether a tool adheres to the CAIS, however, is not as easily determined. Through static analysis of the tool, adherence to Value Limits can be determined since parameter typing information is all that is necessary. In reference to the limit above, a tool that uses channels to communicate can be examined statically to determine that channel names have 20 or fewer characters. Determining whether the same tool adheres to the limit placed on simultaneous channels, however, is not as easy. Since adherence to Use Limits depends on the control structure of the tool, dynamic instrumentation is often necessary. Input from external sources, such as user input or information from other processes, often determines the execution path through a tool such as an editor or command language interpreter. In these cases, static analysis can only indicate which inputs determine control flow.

The form in which limits are specified can have consequences on CAIS validation and tool transportability. The use of defined constants and type attributes in the definition of Ada has eased language validation and increased program transportability. By using much the same specification technique for CAIS limits, CAIS validation can be simplified and tool transportability can be increased. To exemplify how the attribute TYPE'LAST and CAIS implementation defined constants can be used, consider in addition to the limits on channels defined above, the following limit from CAIS Section 5.2.5.2.

> Each element of a direct-access file is selected by an integer index of type COUNT. A conforming implementation must at least support a range of indices from 1 to 32767.

If the CAIS implementation was required to define the attribute COUNT'LAST to indicate the upper bound for the index then the following implications hold. Tool writers could in many applications construct tools whose correct operation depended on the attribute rather than a specific predetermined upper bound. The advantage of writing tools in this manner is that transportability is gained at the cost of performance differences.

To see how constants can also be used to ease validation and increase tool portability, consider the following alternative specification of the channel Use Limit given above.

> Conforming implementations shall define the constant MAXIMUM_SIMULTANEOUS_CHANNELS and support exactly that many simultaneous accepting channels. A minimum value for the constant shall be 20.

Without such a constant, the validator does not know, independent of the implementation, how many accepting channels are implemented. The validation suite must simply be an exerciser. Not knowing what limit is implemented, the validation suite would be unable to expect well defined behavior within and outside the range. To accommodate this, the validation suite could be changed to fit each separate CAIS validated. One set of test cases could address values within the implemented limit and expect acceptable results, and another set of cases could address robust behavior outside the implemented limit. Without the constant, the suite would have to be manually adjusted to each implementation to know which inputs should produce functional results and which should demonstrate robust behavior.

V. SUMMARY

In this paper a specification technique for the Common Ada Programming Support Environment Interface Set (CAIS) has been presented. The specification consists of parts detailing the syntax of operations, Abstract Machine Programs to demonstrate the functionality of operations, Protocols and Hidden Interfaces to indicate interactions among operations, and Pragmatic Limits for implementations. The paper argues that an Abstract Machine based on Ada provides a well-defined description technique for CAIS functionality. The use of an Ada-based Abstract Machine is motivated by CAIS users familiarity with Ada, and the ability to produce a validation tool from the descriptions. The paper shows that Protocols and Hidden Interfaces are necessary to a complete specification, and shows how they can be detailed through commentary and Abstract Programs. Two types of implementation limits are presented as necessary to CAIS (Use Limits and Value Limits), and a format for specifying limits is presented.

The specification technique presented in this paper is being applied to the Process Control and Node Model packages of CAIS by the authors. Further work is currently in progress to show how a validation suite can be generated from an Abstract Machine based specification of CAIS. The approach being taken is to use Abstract Programs and Limits to identify needed test cases. This is done in a white-box (using the instructions in Abstract Programs) fashion. Since validation is to be done at the interface, rather than by observing implementation details, the anticipated results for each test case are constructed from Protocol and Hidden Interface information.

VI. REFERENCES

[1] Castor,V.L. APSE Evaluation and Validation Team Plan. Wright Patterson AFB, Ohio, November 1983.

[2] Draft Specification of the Common APSE Interface Set (CAIS), Version 1.1, Ada Joint Program Office, Pentagon, Washington, D.C., September 1983.

[3] Freedman,R.S., Specifying KAPSE Interface Semantics, in Kernel Ada Programming Support Environment (KAPSE) Interface Team: Public Report Volume II, NOSC TD522, Naval Ocean Systems Center, San Diego CA, pp. 3m1-3m13, October 1982.

[4] Freedman,R.S., The Need for an Operational Semantic Definition of CAIS, Position Statement, Hazeltine Corp, Greenlawn, NY, December 1983.

[5] Goodwin,J.P., A Revised Stoneman for Distributed Ada Support Environments, Technical Report, Department of Computer Science, CS830010, Virginia Tech, Blacksburg VA, October 1983.

[6] Hoare,C.A.R., An Axiomatic Basis for Computer Programming, Communications of ACM, October 1969.

[7] Kafura, Lee, Lindquist, and Probert, Validation in Ada Programming Support Environments, Technical Report Department of Computer Science, CSIE-82-12, Virginia Tech, Blacksburg VA, December 1982.

[8] Oberndorf,P., Ada Programming Support Environments (APSE) Interoperability and Transportability (I&T) Management Plan, in Kernel Ada Programming Support Environment (KAPSE) Interface Team: Public Report Volume III, NOSC TD522, Naval Ocean Systems Center, San Diego CA, pp. 3A1-3A34, June 1983.

[9] Probert,T.H., Ada Validation Organization: Policies and Procedures, Mitre Corporation, Washington D.C., Report MTR-82W00103, June 1982.

[10] Stoy,J., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, Cambridge MA 1977.

TECHNICAL REPORTS DISTRIBUTION LIST

Leader Information Sciences
Engineering Sciences Directorate
Office of Naval Research
800 North Quincy St.
Arlington, VA 22217

Office of Naval Research Resident
Representative, Joseph Henry Building
Room 623
2100 Pennsylvania Avenue, N. W.
Washington, DC 20375

Director, Naval Research Laboratory
ATTN: Code 2627
Washington, DC 20375

Defense Technical Information Center
Building 5, Cameron Station
Alexandria, VA 22314

V. L. Castor
AFWAL/AAAF
Wright-Patterson AFB, Ohio 45433

Dr. Jack Kramer
I.D.A.
1801 N. Beauregard St.
Alexandria, VA 22311

Lt. Cdr Brian Schaar
Ada Joint Program Office
3D139 (400AN)
Pentagon
Washington, DC 20301

Mitch Bassman
Computer Sciences Corp.
6565 Arlington Blvd.
Falls Church, VA 22046
M/C 281

Frank Belz
TRW DSG
One Space Park
R2/1127
Redondo Beach, CA 92078

Tom Conrad
NUSC
Bldg. 1171
Newport, RI 02840

Bob Converse
NAVSEA
PMS-408
Washington, DC 20362

Jay Ferguson
DoD
ATTN: T303, Jay Ferguson
9800 Savage Rd.
Ft. Meade, MD 20755

Jack Foidl
TRW DSG
3420 Kenyon St. *202
San Diego, CA 92110

John Foreman
Texas Instruments, Inc.
P. O. Box 405 M/S 3407
Lewisville, TX 75067

Barbara Fromhold
U.S. Army CECOM
DRSEL-TCS-ADA-3
Ft. Monmouth, NJ 07703

Tim Harrison
Texas Instruments, Inc.
P. O. Box 405 M/S 3407
Lewisville, TX 75067

Hal Hart
TRW DSG
One Space Park
R2/1127
Redondo Beach, CA 92078

Doug Johnson
SoftWrights Inc.
1401 N. Central Expressway
Suite 100
Richardson, TX 75080

Larry Johnston
NADC
Code 503
Warminster, PA 18974

Elizabeth Kean
RADC/COES
Griffiss AFB, NY 13441

Rudolph Krutar
Elizabeth Wald
NRL
Code 515Ø
4555 Overlook Ave., SW
Washington, DC 20375

Bill Laplant
HQ USAF/SITT
Washington, DC 2033Ø

Larry Lindley
NAC D/Ø72.21
6ØØØ E. 21st St.
Indianapolis, IN 46218

Warren Loper
NOSC
Code 8315
San Diego, CA 92152

Lucas M. Maglieri
National Defense Hqds.
1Ø1 Colonel Bay Dr.
Ottawa, Ontario K1AØK2

Jo Miller
NWC
Code 3192
China Lake, CA 93555

Gil Myers
NOSC
Code 8322
San Diego, CA 92152

Philip Myers
Dave Pasterchik
NAVELEX
ELEX 8141A
Washington, DC 2036Ø

MITRE Corp.
K2Ø3
P. O. Box 2Ø8
Bedford, MA Ø173Ø

Eldred Nelson
TRW DSG
One Space Park
R2/1Ø76
Redondo Beach, CA 9Ø278

Tricia Oberndorf
NOSC
Code 8322
San Diego, CA 92152

Shirley Peele
Guy Taylor
FCDSSA
Code 822
Bldg. 1279 Dam Neck
Virginia Beach, VA 23461

Lee Purrier
George Robertson
FCDSSA
Code 822
2ØØ Catalina Blvd.
San Diego, CA 92147

Mo Stein
Ed Dudash
NSWC/DL
Code N31
Dahlgren, VA 22448

Tucker Taft
Jim Moloney
Intermetrics
733 Concord Ave.
Cambridge, MA Ø2138

Rich Thall
SofTech
46Ø Totten Pond Road
Waltham, MA Ø2154

Chuck Waltrip
Johns Hopkins University
Applied Physics Lab
Johns Hopkins Road
Laurel, MD 2Ø7Ø7

Bill Wilder
SofTech
Three Skyline Place
Suite 5ØØ
52Ø1 Leesburg Pike
Falls Church, VA 22Ø41

Bernie Abrams
Charles Mooney
Grumman Aerospace
Mail Station B38-35
Bethpage, NY 11714

Dennis Cornhill
John Beane
Honeywell/SRC
2600 Ridgeway Pkwy.
MN17-2351
Minneapolis, MN 55413

Fred Cox
EES/SEL/DSD
Georgia Tech
Atlanta, GA 30332

Dick Drake
IBM
Federal Systems Division
102/075
Godwin Drive
Manassas, VA 22110

Jon Fellows
System Development Corp.
5151 Camino Ruiz, 02-B14
Camarillo, CA 93010

Herman Fischer
Litton Data Systems
MS 64-30
8000 Woodley Ave.
Van Nuys, CA 91409

Roy Freedman
Hazeltine Corp.
Research Laboratories
Greenlaw, NY 11740

Anthony Gargaro
Computer Sciences Corp.
304 W. Route 38
Moorestown, NY 08057

Steve Glaseman
Teledyne Systems Co.
19601 Nordhoff St.
Northridge, CA 91324

Eric Griesheimer
Nicholas Baker
McDonnel Douglas Astronautics
5301 Bolsa M/S 11-2
Huntington Beach, CA 92647

Ron Johnson
Boeing Aerospace Co.
3903 Hampton Way
Kent, WA 98032

Judy Kerner
Norden Systems M/S M171
P. O. Box 5300
Norwalk, CT 06856

Reed Kotler
Lockheed Missiles & Space
1111 Lockheed Way
Sunnyvale, CA 94086

Pekka Lahtinen
Oy Softplan AB
P. O. Box 209
SF-33100 Tampere 10
Finland

Eli J. Lamb
Bell Labs - 3A 405
600 Mountain Avenue
Murray Hill, NJ 07974

Dave Loveman
Massachusetts Computer
Assoc., Inc.
26 Princess St.
Wakefield, MA 01880

Tim Lyons
Software Sciences Ltd.
Abbey House
Farnborough Hampshire
GU14 7NB
England

Dave McGonagle
GE OR&D K-1
Schenectady, NY 12345

H. R. Morse
Frey Federal Systems
Chestnut Hill Rd.
Amherst, NH 03031

Erhard Ploedereder
c/o Tartan Laboratories
477 Melwood Ave.
Pittsburgh, PA 15213

Ann Reedy
PRC
1500 Planning Res. Dr.
5W1
McLean, VA 22102

Jim Ruby
Hughes Aircraft Co.
P. O. Box 3310, 618/P215
Fullerton, CA 92634

Sabina Saib
General Research Corp.
P. O. Box 6770
Santa Barbara, CA 931111

Edgar Sibley
Alpha Omega Group, Inc.
World Building Suite 406
8121 Georgia Avenue
Silver Spring, MD 20910

Rob Westermann
TNO-IBBC
P. O. Box 9
2600 AA Delft
The Netherlands

Herb Willman
Raytheon Company - MSD
Hartwell Road (GRA-1)
Bedford, MA 01730

Doug Wrege
Dianna Humphrey
Control Data Corp.
5500 Interstate N. Pkwy.
Atlanta, GA 30328

Larry Yelowitz
Ford Aerospace & Communications Corp. WDL
3939 Fabian Way MSV02
Palo Alto, CA 94303

Gina Burt
AFALC/PTEC
Wright-Patterson AFB
OH 45433

Chris Anderson
AFATL/DLMM
Elgin AFB
FL 32542

Bob Harrell
AFCCPC/SKXX
Tinker AFB
OK 73145

Dave Fautheree
AFCMD/KRS
Kirtland AFB
NM 87117

John Prentice
AFHRL/IDC
Lowry AFB
CO 80230

John Taylor
AFLC/MMEC
Wright-Patterson AFB
OH 45433

Rick Long
AFWAL/AAAF-2
Wright-Patterson AFB
OH 45433

Rich Wallace
AFWAL/AAAF-2
Wright-Patterson AFB
OH 45433

Jimmy Williamson
AFWAL/AAAF-2
Wright-Patterson AFB
OH 45433

Mark Mears
AFWAL/FIGLB
Wright-Patterson AFB
OH 45433

Georgeanne Chitwood
ASD/ADOL
Wright-Patterson AFB
OH 45433

Nelson Estes
ASD-AFALC/AXTS
Wright-Patterson AFB
OH 45433

Dan Burton
ESD/ALL
Hanscom AFB, MA 01731

Don Jennings
OC-ALC/MMECE
Tinker AFB, OK 73145

Pat Maher
DO-ALC/MMECF
Hill AFB, UT 84056

Sam Dugan
SA-ALC/MMEC
Kelly AFB, TX 78241

John Miller
SM-ALC/MMEHP
McClellan AFB, CA 31098

Palmer Craig
WR-ALC/MMESM
Robins AFB, GA 31098

Rich Fleming
Aerospace Corp.
M1/112
P. O. Box 92957
Los Angeles, CA 90009

Gregg Bettice
Naval Avionics Ctr.
Code 825
6000 E. 21st St.
Indianapolis, IN 46218

Ronnie Martin
Georgia Institute
of Technology
Atlanta, GA 30332

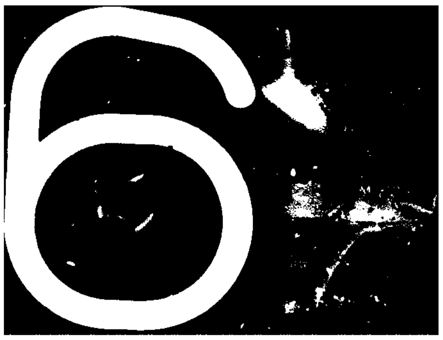
Terry Humphrey
Johnson Space Ctr.
Mail Station EH-4
Houston, TX 77058

Lucas Maslieri
National Defense HQ
101 Colonel By Drive
Ottawa Ontario
K1A OK2

Kevin Chadwick
National Defense HQ
101 Colonel By Drive
Ottawa Ontario
K1A OK2

Bob Knapper
Institute for
Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311

Betsy Kruesi Bailey
Institute for
Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311


6